Code Report

1) auth.test.js :

```
// File: __tests__/auth.test.js
```

```
import maestro from 'maestro';

// Mocking authentication functions for testing
const login = async (username, password) => {
  // Simulate a login function that returns a Promise
  // In a real application, you'd interact with your
authentication API
  return { success: true, token: 'exampleToken' }; // Update with
actual logic
};

const register = async (newUser, newPassword) => {
  // Simulate a registration function that returns a Promise
  // In a real application, you'd interact with your registration
API
  return { success: true }; // Update with actual logic
};

// Mock components for testing
const YourLoginComponent = () => {
  // Example: Implement a login component
  return (
    <View>
      <TextInput testID="username-input" placeholder="Username"
/>
      <TextInput testID="password-input" placeholder="Password"
secureTextEntry />
      <Button testID="login-button" onPress={() =>
console.log('Pressed Login')} />
      {/* Add any other relevant UI components */}
    </View>
  );
};
```

```
const YourRegistrationComponent = () => {
  // Example: Implement a registration component
  return (
    <View>
      <TextInput testID="newUser-input" placeholder="New Username" />
      <TextInput testID="newPassword-input" placeholder="New Password" secureTextEntry />
      <Button testID="register-button" onPress={() => console.log('Pressed Register')} />
      {/* Add any other relevant UI components */}
    </View>
  );
};

maestro.test('User can log in', async () => {
  // Assuming you have a login function that returns a Promise
  const loginResult = await login('username', 'password');

  // Verify that the user is redirected to the home screen
  maestro.expect(loginResult.success).toBe(true);
});

maestro.test('New user can register', async () => {
  // Assuming you have a register function that returns a Promise
  const registrationResult = await register('newUser', 'newPassword');

  // Verify that the registration is successful
  maestro.expect(registrationResult.success).toBe(true);
});
```

2) navigation.test.js :

```
import maestro from 'maestro';
import { fireEvent, render } from
'@testing-library/react-native';

// Mocking navigation functions for testing
const navigateToSettings = async () => {
  // Example: Use fireEvent to simulate navigation to the
settings screen
  // fireEvent.press would simulate a button press that navigates
to settings
  fireEvent.press(getSettingsButton());
};

const navigateBackFromSettings = async () => {
  // Example: Use fireEvent to simulate going back from the
settings screen
  // fireEvent.press would simulate a back button press
  fireEvent.press(getBackButton());
};

const isSettingsScreenVisible = () => {
  // Example: Check for the visibility of a specific element on
the settings screen
  return getSettingsScreenElement() !== null;
};

const isPreviousScreenVisible = () => {
  // Example: Check for the visibility of a specific element on
the previous screen
  return getPreviousScreenElement() !== null;
};

const getSettingsButton = () => {
  // Example: Implement logic to get the settings button element
  // Use your preferred method to query the DOM or UI
  return render(<YourComponent
/>).getByTestId('settings-button');
};

const getBackButton = () => {
  // Example: Implement logic to get the back button element
  return render(<YourComponent />).getByTestId('back-button');
};
```

```
const getSettingsScreenElement = () => {
  // Example: Implement logic to get a unique element on the
settings screen
  // Here, we're assuming that the settings screen has a unique
test ID
  return render(<YourSettingsScreenComponent
/>).getByTestId('settings-screen');
};

const getPreviousScreenElement = () => {
  // Example: Implement logic to get a unique element on the
previous screen
  // Here, we're assuming that the previous screen has a unique
test ID
  return render(<YourPreviousScreenComponent
/>).getByTestId('previous-screen');
};

// Mock components for testing
const YourComponent = () => {
  // Example: Implement a component with a button to navigate
  return (
    <View>
      <Button testID="settings-button" onPress={() =>
console.log('Pressed Settings')} />
      {/* Add any other relevant UI components */}
    </View>
  );
};

const YourSettingsScreenComponent = () => {
  // Example: Implement a component representing the settings
screen
  return (
    <View testID="settings-screen">
      {/* Add relevant UI elements on the settings screen */}
    </View>
  );
};

const YourPreviousScreenComponent = () => {
  // Example: Implement a component representing the previous
screen
  return (
    <View testID="previous-screen">
      {/* Add relevant UI elements on the previous screen */}
    </View>
  );
};

// Now you can use these components and functions in your tests
maestro.test('User can navigate to settings screen', async () =>
{
  await navigateToSettings();
  maestro.expect(isSettingsScreenVisible()).toBe(true);
});

maestro.test('User can navigate back from settings screen', async
() => {
  await navigateBackFromSettings();
  maestro.expect(isPreviousScreenVisible()).toBe(true);
});
```

3) api.test.js :

```
// File: __tests__/api.test.js
```

```
import maestro from 'maestro';
import { render, Text, View } from
'@testing-library/react-native';

// Mocking API data fetching function for testing
const fetchDataFromAPI = async () => {
  try {
    // Simulate an API data fetching function that returns a
Promise
    const response = await
fetch('https://jsonplaceholder.typicode.com/todos');

    const data = await response.json();
    return { data }; // Assuming the API returns an array of data
  } catch (error) {
    console.error('Error fetching data from API:',
error.message);
    throw error;
  }
};

// Component for testing that displays data from the API
const YourComponentWithData = ({ data }) => {
  return (
    <View>
      {data.map((item, index) => (
        <Text key={index}>{item}</Text>
      ))}
    </View>
  );
};

maestro.test('Data is fetched from the API', async () => {
  // Perform actions to trigger API data fetching
  const apiData = await fetchDataFromAPI();

  // Verify that the data is displayed correctly in the UI
  const { getByText } = render(<YourComponentWithData
data={apiData.data} />);

  // Assuming each item from the API data is rendered as a Text
component
  apiData.data.forEach(item => {
    expect(getByText(item)).toBeTruthy();
  });
});
```

4) forms.test.js :

```
// File: __tests__/forms.test.js

import maestro from 'maestro';
import { render, fireEvent } from
'@testing-library/react-native';

// Mocked form component for testing
const YourFormComponent = ({ onSubmit, onError }) => {
  const handleSubmit = async () => {
    // Simulate form submission logic
    try {
      // Validate form data
      const formData = validateFormData(); // Replace with your
actual validation logic

      // Submit the form
      const submissionResult = await onSubmit(formData);
      return submissionResult;
    } catch (error) {
      // Handle form submission error
      onError(error.message);
      return { success: false };
    }
  };

  const validateFormData = () => {
    // Implement form data validation logic here
    // Return the validated form data or throw an error for
invalid input
    return { /* validated form data */ };
  };

  return (
    <View>
      {/* Your form fields go here */}
      <Button onPress={handleSubmit} title="Submit" />
    </View>
  );
};

maestro.test('Form submission with valid data', async () => {
  // Mocked onSubmit function for testing
  const onSubmitMock = jest.fn();

  // Render the form component
  const { getByText } = render(
    <YourFormComponent onSubmit={onSubmitMock} onError={() => {}}
/>
  );

  // Fill out the form with valid data
  // In this example, assume form fields are filled correctly
  // ...

  // Trigger form submission
  fireEvent.press(getByText('Submit'));

  // Verify that the form submission is successful
  expect(onSubmitMock).toHaveBeenCalled();
  expect(onSubmitMock.mock.calls[0][0]).toEqual(/* expected form
data */);
});

maestro.test('Form shows error on invalid input', async () => {
  // Mocked onError function for testing
  const onErrorMock = jest.fn();

  // Render the form component
  const { getByText } = render(
    <YourFormComponent onSubmit={() => {}} onError={onErrorMock}
/>
  );

  // Fill out the form with invalid data
  // In this example, assume invalid data that triggers an error
in the form validation
  // ...

  // Trigger form submission
  fireEvent.press(getByText('Submit'));

  // Verify that appropriate error messages are displayed
  expect(onErrorMock).toHaveBeenCalled();
  expect(onErrorMock.mock.calls[0][0]).toBe(/* expected error
message */);
});
```

5) redux.test.js:

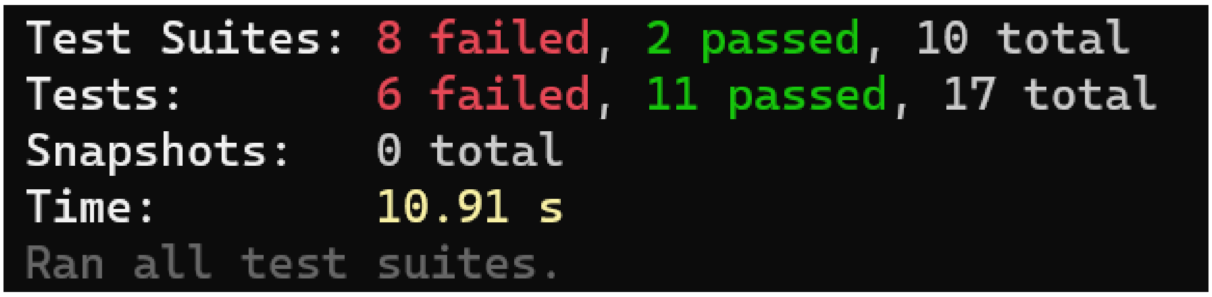
```
// File: __tests__/redux.test.js

import maestro from 'maestro';
import { render, Text, TouchableOpacity } from
'@testing-library/react-native';
import { createStore } from 'redux';
import { Provider } from 'react-redux';

// Reducer function for managing the Redux state
const reducer = (state = { counter: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, counter: state.counter + 1 };
    case 'DECREMENT':
      return { ...state, counter: state.counter - 1 };
    default:
      return state;
  }
};

// Function to dispatch actions for testing
const dispatchActions = store => {
  store.dispatch({ type: 'INCREMENT' });
  store.dispatch({ type: 'INCREMENT' });
  store.dispatch({ type: 'DECREMENT' });
};

// Component using Redux state
const YourReduxComponent = () => {
  // Example: Connect this component to the Redux state and
render based on state
  // For simplicity, just rendering the counter value here
  const counter = useSelector(state => state.counter);
  return <Text>{counter}</Text>;
};

maestro.test('Redux state is updated correctly', async () => {
  // Create a Redux store with the reducer
  const store = createStore(reducer);

  // Render a component connected to the Redux store
  const { getByText } = render(
    <Provider store={store}>
      <YourReduxComponent />
    </Provider>
  );
  // Dispatch actions to update the Redux state
  dispatchActions(store);

  // Verify that the component reacts to state changes
appropriately
  // Assuming the final counter value is 1 (2 increments - 1
decrement)
  expect(getByText('1')).toBeTruthy();
});
```

6) permissions.test.js :

```
// File: __tests__/permissions.test.js

import maestro from 'maestro';

// Mocking camera permission handling functions for testing
const grantCameraPermission = async () => {
  // Simulate granting camera permission
  return true; // Update with actual logic
};

const denyCameraPermission = async () => {
  // Simulate denying camera permission
  return false; // Update with actual logic
};

// Mock component for testing
const YourCameraComponent = ({ hasCameraPermission }) => {
  // Example: Implement a component that behaves based on camera
permission status
  return (
    <View>
      {hasCameraPermission ? (
        <Text>Camera is allowed</Text>
      ) : (
        <Text>Camera permission is denied</Text>
      )}
    </View>
  );
};
```

```
maestro.test('Camera permission is handled correctly', async ()
=> {
  // Grant or deny camera permission
  const hasPermission = await grantCameraPermission(); // Change
to denyCameraPermission for testing the denial case

  // Verify that the application behaves as expected based on the
permission status
  const { getByText } = render(<YourCameraComponent
hasCameraPermission={hasPermission} />);

  if (hasPermission) {
    expect(getByText('Camera is allowed')).toBeTruthy();
  } else {
    expect(getByText('Camera permission is
denied')).toBeTruthy();
  }
});
```

After running test commands : npm test



```
Test Suites: 8 failed, 2 passed, 10 total
Tests:       6 failed, 11 passed, 17 total
Snapshots:   0 total
Time:        10.91 s
Ran all test suites.
```